# A Delay-Insensitive Multiply-Accumulate Unit

Christian D. Nielsen
Alain J. Martin

Computer Science Department
California Institute of Technology

Caltech-CS-TR-92-03

| | |
|---|---|
| # Report Documentation Page | *Form Approved*<br>*OMB No. 0704-0188* |

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

| | | |
|---|---|---|
| 1. REPORT DATE<br>**12 FEB 1992** | 2. REPORT TYPE | 3. DATES COVERED<br>**12-02-1992 to 12-02-1992** |

| | |
|---|---|
| 4. TITLE AND SUBTITLE<br>**A Delay-Insensitive Multiply-Accumulate Unit** | 5a. CONTRACT NUMBER |
| | 5b. GRANT NUMBER |
| | 5c. PROGRAM ELEMENT NUMBER |
| 6. AUTHOR(S) | 5d. PROJECT NUMBER |
| | 5e. TASK NUMBER |
| | 5f. WORK UNIT NUMBER |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)<br>**Defense Advanced Research Projects Agency,3701 North Fairfax Drive,Arlington,VA,22203-1714** | 8. PERFORMING ORGANIZATION REPORT NUMBER |
| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) | 10. SPONSOR/MONITOR'S ACRONYM(S) |
| | 11. SPONSOR/MONITOR'S REPORT NUMBER(S) |

12. DISTRIBUTION/AVAILABILITY STATEMENT

**Approved for public release; distribution unlimited**

13. SUPPLEMENTARY NOTES

14. ABSTRACT

**see report**

15. SUBJECT TERMS

| | | | | | |
|---|---|---|---|---|---|
| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES<br>**21** | 19a. NAME OF RESPONSIBLE PERSON |
| a. REPORT<br>**unclassified** | b. ABSTRACT<br>**unclassified** | c. THIS PAGE<br>**unclassified** | | | |

# A Delay-Insensitive Multiply-Accumulate Unit[*]

Christian D. Nielsen[†] and Alain J. Martin
Department of Computer Science
California Institute of Technology
Pasadena, CA91125

February 12, 1992

## 1 Introduction

Due to advances in integration technology the use of asynchronous circuits has become increasingly interesting. Design methods have emerged with which it is manageable to design efficient and reliable asynchronous circuits.

Instead of designing circuits under worst case assumptions as for synchronous circuits, the objective in asynchronous design is to attain the best possible *average* performance and to utilize this potential performance advantage at the architectural level.

We have designed a serial-parallel multiply-accumulate unit that exploits this performance advantage. The unit is designed to be part of a large ring network of units performing vector-matrix multiplications. As the system contains a large number of these multiply-accumulate units, we choose the area-economic serial-parallel approach. Further we want the design to take advantage of the fact that a large percentage of the elements in the matrix are small integers, with zero as a special case. The result is a flexible multiply-accumulator with performance proportional to the bit length of the serial input multiplier.

The design has been implemented as a delay-insensitive circuit, i.e. the functional correctness is independent of any delays in circuit elements as well as wires — except for certain wire forks, called isochronic forks, for which we assume that the difference in delays in the branches of the fork are negligible [4]. This kind of circuits constitutes a sub-class of the class of asynchronous circuits.

The paper describes the design and implementation of the multiply-accumulate unit using the method and tools developed at Caltech [4] for design

[*] The research described in this report was sponsored by the Defense Advanced Research Projects Agency, DARPA Order number 6202; and monitored by the Office of Naval Research under contract number N00014-87-K-0745.

[†] On leave from Department of Computer Science, Technical University of Denmark.

of delay-insensitive circuits. With the use of the method, which consists of a sequence of transformations to a circuit description, the designer goes through the following steps:

Algorithm
⇓
CSP-specification
⇓
Handshake expansion
⇓
Production Rules
⇓
Layout

The transformations, which are performed at each level, are supported by interactive design and analysis tools from the handshake expansion level down.

The purpose of this paper is twofold: To present a delay-insensitive serial-parallel multiplier and to illustrate the full course of design from a high-level description to fabrication on a non-trivial example.

The description of the method should be seen as an attempt to give the reader insight in the design of a full scale delay-insensitive circuit from top to bottom; it is not a complete presentation of the design method (for this we refer to [4]).

We have fabricated an eight-bit prototype of the multiply-accumulate unit in $2\mu$ CMOS. Because of the delay-insensitivity the chip is very robust towards variations in operating conditions. At room temperature and 5 volts the chip has a cycle time of 27 nsec for a one bit serial-parallel multiplication. The design is scalable to wider word sizes without loss of performance.

## 2 Algorithm → CSP-specification

The algorithm is inspired by previous work on a digital artificial neural network engine [6]. The architecture of this network is based on a systolic ring network proposed by Kung and Hwang [2]. In this architecture the neural computations are performed as consecutive vector-matrix multiplications, where the vector represents the state of the neural network and each element in the matrix represents the weight of a connection between two neuron processors. A zero weight represents "no connection".

So, the core of the neural computation is to perform a vector-matrix multiplication:

$$\underline{P} = \underline{\underline{A}} \times \underline{B} \quad \text{or} \quad \begin{pmatrix} P_1 \\ P_2 \\ \vdots \\ P_N \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1N} \\ A_{21} & A_{22} & \cdots & A_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ A_{N1} & A_{N2} & \cdots & A_{NN} \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_N \end{pmatrix} \tag{1}$$

This multiplication is performed by arranging a set of multiply-accumulate processors in a ring network. To each processor is attached a local memory containing one row of the matrix. The vector elements are distributed in the ring — one for each processor — and circulated among the processors during operation. The task for each processor is to calculate one inner product of the result vector, i.e. for processor $i$:

$$P_i = \sum_{j=1}^{N} A_{ij} B_j \tag{2}$$

We have chosen the serial-parallel approach for the implementation of this computation for the following reason: Assuming bit lengths, m and n, of the multiplicand and the multiplier respectively, the size of the iterative multiplier is $O(m+n)$ in contrast to $O(mn)$ for a combinational counterpart. This area consideration becomes of interest already for bit lengths of ten to twenty. The final system of neuron processors will consist of many hundreds of identical processors in which the multiplier is a principal component. The size of the multiplier does therefore greatly influence the size of the full system.

The summation in (2) is expanded to a serial-parallel implementation:

$$P_i = \sum_{j=1}^{N} \sum_{k=0}^{n-1} 2^k a_{ijk} B_j \quad , \tag{3}$$

where $A_{ij} = \sum_{k=0}^{n-1} 2^k a_{ijk}$.

From this formula we can formulate a CSP-specification for processor $i$, $1 \le i \le N$:

```
$PROC[i] \equiv (sum := 0;\ j := 1;$
        *[ $j \le N \longrightarrow k := 0;$
                   *[ $k < n \longrightarrow sum := sum + 2^k a_{ijk} B_j;$
                                 $k := k + 1$
                   ];
                   $j := j + 1$
        ];
        $P_i := sum$
       )
```

Next, we decompose the processor specification into a process controlling the loop indices, $j$ and $k$, (called "*ENVIRONMENT*") and a process performing the computation (called "*MAP*"):

```
$ENVIRONMENT \equiv$ ( $P?P_i$; $j := 1$;
                    *[ $j \leq N \longrightarrow B!B_j$;
                                      $k := 0$;
                                      *[ $k < n \longrightarrow A!a_{ijk}$;
                                                          $k := k + 1$
                                       ];
                                      $j := j + 1$
                     ];
                    $P?P_i$
                   )
```

```
$MAP \equiv$ *[[ $\overline{B} \longrightarrow B?bb$
           [] $\overline{A} \longrightarrow A?a'$;  $sum := sum + a \cdot bb$;  $bb := 2 \cdot bb$
           [] $\overline{P} \longrightarrow P!sum$;  $sum := 0$
          ]]
```

All internal variables in *MAP* except $a'$ are integers; $a'$ represents a single bit of the multiplier $A$. The initialization of *sum* to zero is done by an initial communication on channel $P$. It is noted that the multiplication with $2^k$ in *MAP* is performed by multiplying *bb* with 2 after each accumulation.

As the two variables, *sum* and *bb*, occur in the expressions that are assigned to themselves, it is necessary to introduce extra variables to hold the values during the assignment:

```
$MAP \equiv$ *[[ $\overline{B} \longrightarrow B?bb$
           [] $\overline{A} \longrightarrow A?a', b := bb, acc := sum$;  $sum := acc + a' \cdot b, bb := 2 \cdot b$
           [] $\overline{P} \longrightarrow P!sum$;  $sum := 0$
          ]]
```

This specification of *MAP* is now in a form, that can be implemented, but first we discuss a possible implementation of the environment.

## The environment

Even though our focus is on the implementation of the multiply-accumulate unit it is necessary to consider how the process constituting the immediate environment for the unit may be implemented. For simplicity we want to avoid using counters to keep track of the loop indices. Furthermore the control of the loop indices is local to each processor, whereas the values of the multiplicand, $B_j$, and the inner product, $P_i$, are send from and to the main ring of processors.

This suggests that the control of the accumulator be stored together with the multiplier bits in the local memory.

With this scheme we are able to utilize variable bit length multipliers. The multiplication may be interrupted and the next started when the remaining most significant bits in the multiplier are all zero. In this way the actual computation time for a multiplication is between zero (for multiplication with zero) and $n$ cycles. For an even distribution of numbers the average time is $n - 1$. For many applications in artificial neural networks, the distribution is not even, but with a bias towards small numbers. Especially for a large class of neural net configurations more than half of the numbers in the matrix will be zero. It is shown in [7] that an asynchronous implementation is well suited to take advantage of these properties.

A possible description of the environment, where the control of the accumulator and the communication of the multiplicands and inner products are separated, is:

$$
\begin{array}{l}
*[\;\mathit{MEM}?w;\;\;[w \leq \mathit{1} \longrightarrow A!w \;\square\; w = \mathit{2} \longrightarrow \mathit{NB} \;\square\; w = \mathit{3} \longrightarrow R\;]] \\
\| \\
*[\;\mathit{RINGI}?s;\;\;[\overline{B} \longrightarrow B!s \;\square\; \overline{Q} \longrightarrow Q?s];\;\; \mathit{RINGO}!s\;] \\
\| \\
*[\;P?x \bullet Q!f(x)\;]
\end{array}
$$

From here, we concentrate on the implementation of $MAP$. We change the specification of $MAP$ to reflect that the control of and communication on the $B$ and $P$ channels are separated:

$$
\begin{array}{rl}
\mathit{MAP} \equiv *[[ & \overline{\mathit{NB}} \longrightarrow \mathit{NB} \bullet B?bb \\
\square & \overline{A} \;\longrightarrow A?a', b := bb, acc := sum;\;\; sum := acc + a' \cdot b, bb := \mathit{2} \cdot b \\
\square & \overline{R} \;\longrightarrow R \bullet P!sum;\;\; sum := \mathit{0} \\
]] &
\end{array}
$$

The environment and accumulator with communication channels are sketched in Figure 1. Note that the construction of the first environment process guarantees that the three guards in $MAP$ are mutually exclusive.

## Carry-save versus ripple-carry adder

Before we decompose our specification, we should decide whether the multiplier should be implemented with a carry-save or ripple-carry adder. The ripple carry adder has become a popular example illustrating the benefits of asynchronous circuits. This is due to the fact that with very simple hardware we obtain very good average performance — logarithmic to the number of bits added [5]. This is a very important property, when the result is needed immediately.

For this application, we do not need the result of each addition in binary form, but only the final inner product. By using a carry-save adder we can do an addition in unit time. The carry part of the accumulated sum needs only to be resolved once per vector-matrix multiplication.

## Computation time

The cycle times associated with input of a new multiplicand, $B_j$, multiplication with an $A$-bit and output of result $P$ are denoted $t_b$, $t_a$, and $t_p$, respectively. The multiplication time for an $n$ bit multiplier is $t_{mult,n} = t_b + n\,t_a$. Multiplication with zero takes time $t_{mult,0} = t_b$. The average multiplication time is $t_{mult} = t_b + \alpha n\,t_a$, where $\alpha n$ denotes the average multiplier bit length for the application. The average time used for the whole task of calculating the inner product is $t_{ip} = N(t_b + \alpha n\,t_a) + (s-1)t_a + t_p$. Here, $s$ is the bit length of the inner product; the term $(s-1)t_a$ is the time needed to flush the carry part into the sum part of the inner product. This is done by repeatedly multiplication with zero bits.

By considering the frequency with which each of the three guarded commands will be activated during operation it is possible to optimize the implementation. The *NB* guard will be activated once per multiplication. During each multiplication the *A* guard will be activated once for each bit in the multiplier. This corresponds to the inner loop of the *ENVIRONMENT* process in section 2. Finally the *R* guard will only be activated once after each full vector-matrix multiplication.

Even if most multipliers are small numbers, the *A* guard is the most frequently activated guard. Hence, the performance of this guarded command should be optimized as much as possible — if necessary at the expense of the others.

## Decomposition

The CSP-specification, *MAP*, is decomposed into processes handling single bit variables only.

The implementation of *MAP* contains at least $m + n - 1$ processes, $MA[l]$, $0 \leq l$, each containing a full-adder:

$$
\begin{array}{lll}
MA[l] \equiv *[\![ & \overline{NB} \longrightarrow NB \bullet B?bb & \\
 & [\!] \; \overline{A} \longrightarrow A?a,\;\; b := bb,\;\; CI?c,\;\; CO!carry,\;\; acc := sum; & \\
 & \qquad sum := SUM(a,b,c,acc),\;\; carry := CARRY(a,b,c,acc), & \\
 & \qquad BI?bb,\;\; BO!b & \\
 & [\!] \; \overline{R} \longrightarrow R \bullet P!sum;\;\; sum := 0 & \\
 & ]\!] &
\end{array}
$$

All variables in the specification are booleans. The value of $B_j$ (with zeros concatenated as most significant bits) is distributed with one bit to each process, i.e. process $MA[l]$ receives $b_{jl}$. The processes further produce each one bit of the accumulated sum, $P_i$. The value of $a_{ijk}$ is send to all processes in parallel. The $BO$ and $CO$ ports in each process is connected to the $BI$ and $CI$ ports of the next more significant bit process.

( To accommodate accumulated sums larger than the minimum limit, the multiply-accumulate processor is extended with an appropriate number of processes. For the sake of regularity, these may be chosen identical to the first $m+n-1$, but they can be simplified to contain half-adders, as they are only accumulating overflowing carries from the accumulation of products. The number of necessary additional processes depends on the application. )

For reasons of efficiency we want to move the communication of the $B$-bits to happen in parallel with the other communications in the $A$-guarded command. This will improve the performance as the unit will perform one communication step followed by an internal step, instead of two communication steps. To make this possible, it is necessary to send the value of $B_j$ shifted one bit to the left, i.e. process $MA[l]$ receives $b_{j(l+1)}$, see Figure 2. Process $MA[l]$ becomes:

```
$MA[l] \equiv *[[\ \overline{NB} \longrightarrow NB \bullet B?bb$
          $[]\ \overline{A} \longrightarrow A?a,\ BI?b,\ BO!bb,\ CI?c,\ CO!carry,\ acc := sum;$
                  $sum := SUM(a,b,c,acc),\ \ carry := CARRY(a,b,c,acc),$
                  $bb := b$
          $[]\ \overline{R} \longrightarrow R \bullet P!sum;\ \ sum := 0$
         $]]$
```

The two ends of the string of processes need to be closed appropriately. In the accumulate cycle ($\overline{A}$ becomes true) process $MA[l]$ starts out by reading in a $b$ and a $c$ from $MA[l-1]$ and sending a $bb$ and a $carry$ to $MA[l+1]$. Process $MA[0]$ communicates with process $MA[-1]$:

```
$MA[-1] \equiv *[[\ \overline{NB} \longrightarrow NB \bullet B?bb$
            $[]\ \overline{A} \longrightarrow A,\ \ BO!bb,\ \ CO!0;\ \ bb := 0$
            $[]\ \overline{R} \longrightarrow R$
           $]]$
```

The process handling the most significant bits, $MA[M]$, $M \geq m+n-1$ is:

```
$MA[M] \equiv *[[\ \overline{NB} \longrightarrow NB$
            $[]\ \overline{A} \longrightarrow A,\ \ BI?,\ \ CI?c,\ \ acc := sum;$
                    $sum := EXOR(c, acc)$
            $[]\ \overline{R} \longrightarrow R \bullet P!sum;\ \ sum := 0$
           $]]$
```

In the following, we will concentrate on the intermediate processes only.

# 3 CSP → Handshake expansion

The implementation of the given specification into production rules can follow one of the two strategies.

- The full handshake expansion is derived directly from the specification including communication actions as well as assignments. Unfortunately the handshake expansion may become very extensive if the combinational expressions are complicated.
- The assignments and message communications are decomposed into separate processes, the data path, which are treated separately. The handshake expansion is then derived from a CSP description including communication actions only; the control part. This yields the possibility to optimize each of the parts separately [3].

Because of the complexity of the boolean expressions for the $SUM$ and $CARRY$ functions, the second strategy is used.

In the following we derive a handshake expansion for $MA[l]$ through the following steps:

1. Separation of the data path from the control part.
2. Replacement of each communication action with its implementation as elementary actions on the two boolean signals that constitute the channel.
3. Reshuffling of actions to optimize performance.

The final transformations of the handshake expansions before implementation as production rules will be treated in the following section.

## Decomposition of data path

From the CSP specification, we derive a specification of the control part by the following steps:

- All message passing is removed from the communication actions.
- All assignments are removed and put into separate processes, leaving each assignment as a simple communication action in the specification, i.e. $*[\ldots; x := y; \ldots]$ becomes $*[\ldots; D; \ldots] \,\|\, *[[\overline{D} \rightarrow x := y; D]]$.
- All communication ports are assigned to be active or passive. We have used the approach that probed ports and output ports are passive and ports corresponding to assignments and input ports (which are not probed) are (lazy) active. The choices are indicated with indices "$P$" or "$A$".

After these steps we get (The active end of the channels are indicated with dots.):

```
$MA[l] \equiv *[[\ \overline{NB} \longrightarrow NB_P \bullet B_A$
        $[]\ \overline{A} \longrightarrow A_P,\ BI_A,\ BO_P,\ CI_A,\ CO_P,\ T_A;\ S_A,\ C_A,\ BB_A$
        $[]\ \overline{R} \longrightarrow R_P \bullet P_P;\ Z_A$
       $]]$
```

The communication actions $T$, $S$, $C$, and $B$ call the data path processes:

```
$*[[\overline{T} \longrightarrow acc := sum;\ \ T_P]]\ \ ||$
$*[[\overline{S} \longrightarrow sum := SUM(a,b,c,acc);\ \ S_P]]\ \ ||$
$*[[\overline{C} \longrightarrow carry := CARRY(a,b,c,acc);\ \ C_P]]\ \ ||$
$*[[\overline{BB} \longrightarrow bb := b;\ \ BB_P]]\ \ ||$
$*[[\overline{Z} \longrightarrow sum := 0;\ \ Z_P]]$
```

## Implementation of communication actions

Each communication action is replaced with its implementation as elementary actions on two boolean signals, that constitutes the channel, e.g. $x_i$ and $x_o$ for communication $X$. We chose from the three four-phase implementations:

Active: $x_o\uparrow; [x_i]; x_o\downarrow; [\neg x_i];$
Lazy active: $[\neg x_i]; x_o\uparrow; [x_i]; x_o\downarrow;$
Passive: $[x_i]; x_o\uparrow; [\neg x_i]; x_o\downarrow;$

This step yields the full handshake expansion:

```
$MA[l] \equiv$
$*[[\ nb_i \longrightarrow [\neg b_i];\ b_o\uparrow, nb_o\uparrow;\ [\neg nb_i \wedge b_i];\ b_o\downarrow, nb_o\downarrow$
  $[]\ a_i \longrightarrow (a_o\uparrow;\ [\neg a_i];\ a_o\downarrow), ([\neg bi_i];\ bi_o\uparrow;\ [bi_i];\ bi_o\downarrow),$
          $([\neg ci_i];\ ci_o\uparrow;\ [ci_i];\ ci_o\downarrow), ([\neg t_i];\ t_o\uparrow;\ [t_i];\ t_o\downarrow),$
          $([bo_i];\ bo_o\uparrow;\ [\neg bo_i];\ bo_o\downarrow), ([co_i];\ co_o\uparrow;\ [\neg co_i];\ co_o\downarrow);$
          $([\neg s_i];\ s_o\uparrow;\ [s_i];\ s_o\downarrow), ([\neg c_i];\ c_o\uparrow;\ [c_i];\ c_o\downarrow),$
          $([\neg bb_i];\ bb_o\uparrow;\ [bb_i];\ bb_o\downarrow)$
  $[]\ r_i \longrightarrow [p_i];\ p_o\uparrow, r_o\uparrow;\ [\neg r_i \wedge \neg p_i];\ p_o\downarrow, r_o\downarrow;\ [\neg z_i];\ z_o\uparrow;\ [z_i];\ z_o\downarrow$
 $]]$
```

## Reshuffling of communication actions

Reshuffling is the process of reorganizing a sequence of actions in a handshake expansion without changing the functionality of the sequence. Reshuffling is primarily applied for performance reasons but it may also add to the simplicity of the hardware. The transformation is essential to the sequencing of events in different processes and does as such influence the performance of the implementation the most. It is performed by the designer, but approximate performance figures may easily be extracted by a cycle analysis tool for comparisons [1].

The $NB$- and $R$-guarded commands are not reshuffled. The first is as simple as it can be. For the $R$-guarded command one might be tempted to postpone the actions, $p_o\downarrow$ and $r_o\downarrow$ to happen in parallel with $z_o\downarrow$, but this will change the functionality of the specification: The value of *sum* (from the CSP-specification) would be reset *while* it is send to the environment.

In each step of the $A$ guarded command the communication actions are specified to happen in parallel, independent of each other. Through analysis of the behavior of several interconnected processes the natural overlapping of the communication actions may be established. It is appreciated that the complexity of the circuit will decrease and the performance increase if it is implemented to perform this natural order of communication actions only. This means that we implement a stronger specification than the parallel operator. Care should be taken that deadlocks are not introduced as communications to the left and the right neighbors overlap.

$$\begin{array}{ll} [\ a_i \longrightarrow & [\neg bi_i \wedge \neg ci_i \wedge \neg t_i];\ \ bi_o\uparrow, ci_o\uparrow, t_o\uparrow, a_o\uparrow;\ \ [bo_i \wedge co_i];\ \ bo_o\uparrow, co_o\uparrow; \\ & [bi_i \wedge ci_i \wedge t_i \wedge \neg a_i];\ \ bi_o\downarrow, ci_o\downarrow, t_o\downarrow, a_o\downarrow;\ \ [\neg bo_i \wedge \neg co_i];\ \ bo_o\downarrow, co_o\downarrow; \\ & [\neg s_i \wedge \neg c_i \wedge \neg bb_i];\ \ s_o\uparrow, c_o\uparrow, bb_o\uparrow;\ \ [s_i \wedge c_i \wedge bb_i];\ \ s_o\downarrow, c_o\downarrow, bb_o\downarrow\ ] \end{array}$$

Further simplification is achieved by collecting signals which change with the same dependencies in one signal. We collect the signals for $BI_A$, $CI_A$ and $T_A$ in $I_A$; $BO_P$ and $CO_P$ in $O_P$; and $S_A$, $C_A$, and $BB_A$ in $X_A$. These collections will be implemented in the section about production rules.
The handshake expansion for the $A$-guarded command is now:

$$\begin{array}{ll} [\ a_i \longrightarrow & [\neg i_i];\ \ i_o\uparrow, a_o\uparrow;\ \ [o_i];\ \ o_o\uparrow;\ \ [i_i \wedge \neg a_i];\ \ i_o\downarrow, a_o\downarrow;\ \ [\neg o_i];\ \ o_o\downarrow; \\ & [\neg x_i];\ \ x_o\uparrow;\ \ [x_i];\ \ x_o\downarrow\ ] \end{array}$$

# 4 Handshake expansion → production rule set

Before the handshake expansion is ready to be implemented as a production rule set we need to perform a couple of transformations; *state assignment* to ensure correct sequencing and *guard strengthening* to ensure non-overlap of guarded commands. These steps are supported by interactive tools.

The production rule set can be implemented directly after these steps.

## State assignment

When we implement the handshake expansion as a production rule set we give up the notion of sequencing using the ";" operator. The sequencing of the production rules needs to be specified explicitly using variables from the handshake expansion only. Therefore, it is necessary that all states in the handshake expansion can be distinguished from each other. If this is not already the case, specific state variables must be inserted.

The problem arises in both the $A$- and $R$-guarded commands. The initial state cannot be distinguished from neither the point after $o_o\downarrow$ nor the point after $p_o\downarrow, r_o\downarrow$, so a state variable should be inserted in each of the two guarded commands. Several heuristics exists for the placement and the performance analysis tools gives guidance to an optimal placement. Generally an optimal place to change a state variable is just before a wait for an input signal transition, but several iterations may be (and was) necessary, as implementation issues at lower levels may play a role.

## Guard strengthening

Finally it is necessary to ensure that the three guarded commands cannot overlap. This may be achieved either by strengthening the guards to ensure that the other commands are finished, or by utilizing that communications on $NB$, $A$, and $R$ are mutually exclusive.

We do not want to strengthen the $A$-guard as it will decrease its performance. Instead we make sure the $NB$ and $R$ channels are "held" as long as their corresponding operations are performed. We "hold" the $NB$ and $R$ channels by making the completion of the communications be the last action in the guarded commands. This is true for the $NB$ communication, but it is necessary to move the completion of the $R$ communication to happen simultaneous with the $Z$ action instead of the $P$ communication.

A similar arrangement for the $A$ channel will cause a performance reduction as it will leave less time for the environment to fetch the next bit. Instead we strengthen the other guards to wait for the $A$-guarded command to finish.

The handshake expansion of $MA[l]$ with necessary state variables and guard strengthenings is:

$$\begin{array}{llll}
\multicolumn{4}{l}{l\text{: } [0, m+n-1]\text{:}} \\
MA[l] \equiv *[[ & nb_i \wedge \neg u \wedge \neg s_o & \longrightarrow & [\neg b_i];\ b_o\uparrow, nb_o\uparrow;\ [\neg nb_i \wedge b_i];\ b_o\downarrow, nb_o\downarrow \\
\quad\square & a_i & \longrightarrow & [\neg i_i];\ i_o\uparrow, a_o\uparrow;\ [o_i];\ o_o\uparrow;\ u\uparrow; \\
 & & & [i_i \wedge \neg a_i \wedge u];\ i_o\downarrow, a_o\downarrow;\ [\neg o_i];\ o_o\downarrow; \\
 & & & [\neg x_i];\ x_o\uparrow;\ u\downarrow;\ [x_i \wedge \neg u];\ s_o\downarrow \\
\quad\square & r_i \wedge \neg u \wedge \neg s_o & \longrightarrow & [p_i];\ p_o\uparrow;\ v\uparrow;\ [v];\ r_o\uparrow;\ [\neg p_i];\ p_o\downarrow; \\
 & & & [\neg z_i];\ z_o\uparrow;\ v\downarrow;\ [\neg r_i \wedge z_i \wedge \neg v];\ z_o\downarrow, r_o\downarrow \\
\quad ]] & & &
\end{array}$$

### Production rule generation

The generation of production rules from the final handshake expansion is straightforward. For each signal transition it is examined, which variables uniquely determines a precondition for the transition. The state assignment described earlier guaranties that this is possible. We get the production rule set for the control circuitry:

Choice $NB$:

$$\begin{array}{lcl}
nb_i \land \neg u \land \neg x_o \land \neg b_i & \rightarrow & b_o\uparrow, nb_o\uparrow \\
\neg nb_i \land b_i & \rightarrow & b_o\downarrow, nb_o\downarrow
\end{array}$$

Choice $A$:

$$\begin{array}{lcl}
a_i \land \neg i_i \land \neg u \land \neg x_o & \rightarrow & i_o\uparrow, a_o\uparrow \\
i_o \land o_i & \rightarrow & o_o\uparrow \\
o_o & \rightarrow & u\uparrow \\
\neg a_i \land i_i \land u & \rightarrow & i_o\downarrow, a_o\downarrow \\
\neg i_o \land \neg o_i & \rightarrow & o_o\downarrow \\
\neg o_o \land u \land \neg x_i & \rightarrow & x_o\uparrow \\
x_o & \rightarrow & u\downarrow \\
\neg u \land x_i & \rightarrow & x_o\downarrow
\end{array}$$

Choice $R$:

$$\begin{array}{lcl}
r_i \land \neg r_o \land \neg u \land \neg x_o \land p_i & \rightarrow & p_o\uparrow \\
p_o & \rightarrow & v\uparrow \\
v & \rightarrow & r_o\uparrow \\
r_o \land \neg p_i & \rightarrow & p_o\downarrow \\
\neg p_o \land v \land \neg z_i & \rightarrow & z_o\uparrow \\
z_o & \rightarrow & v\downarrow \\
\neg r_i \land \neg v \land z_i & \rightarrow & z_o\downarrow \\
\neg v \land \neg z_o & \rightarrow & r_o\downarrow
\end{array}$$

## 5 Production rule set → layout

The production rule set needs to go through a series of transformations before it is ready to be implemented in layout.

### Bubble reshuffling

For a production rule to be implemented in CMOS we need to impose some restrictions on the *polarity* of the signals in a production rule:

- All variables in the guard of an up transition must appear in inverted form, e.g. $\neg x \rightarrow y\uparrow$.

- All variables in the guard of a down transition must appear in true form, e.g. $x \longrightarrow y\downarrow$.

These restrictions arise from electrical properties of the p- and n-transistors in the CMOS technology.

It is necessary to go through each set of production rules and change the polarity of variables in order to make them meet the criteria. This process, called bubble reshuffling, must take into account that internal signals may contain isochronic forks, i.e. forks where the difference in the delays of the branches are assumed be negligible. It is not allowed to invert only one branch of these forks. Production rule sets occur which cannot be resolved in this manner, because of a cyclic dependency between the variables. In these cases it is necessary to go back to the handshake expansion and reshuffle the troublesome events.

The whole procedure of bubble reshuffling is automated with performance analysis. In the cases of cyclic dependencies the particular variables are pointed out.

### Transistor sizing

Transistors are sized in order to increase performance. We concentrate primarily on the sizing of the production rules for the $A$-guarded command. Further we keep the load from the other guarded commands on the shared signals, $x_o$ and $u$, small.

Given bounds on smallest, average and largest transistor width the transistor sizing is performed automatically to repeatedly optimize the critical cycle of events in the circuit [1].

### Layout

Layout is automatically produced from the final sized production rule set. The up and down transitions of each variable are collected in a cell. If the cell is not combinational a "staticizer"(a weak feedback loop) is added to the output. The cells are automaticly placed and routed, see Figure 3. The total transistor count for the control part is 129.

## 6 Data Path

The production rules and transistor network for the combinational expressions and the communication ports have been designed by hand. We have used the standard communication ports as they have been derived in [3]. All bit variables are represented in dual rail. At an input port the dual rail variable is input into a register, which content is stable at the time of use. This requires that the register acknowledges when the variable has been input, Figure 4:

$$\begin{array}{lcl@{\qquad\qquad}lcl} xt & \rightarrow & x0\downarrow & xf & \rightarrow & x1\downarrow \\ \neg x0 & \rightarrow & x1\uparrow & \neg x1 & \rightarrow & x0\uparrow \end{array}$$

$$\begin{array}{lcll} x0 & \rightarrow & x1\downarrow & (\textit{feedback to staticize signal}\,) \\ x1 & \rightarrow & x0\downarrow & (\textit{feedback to staticize signal}\,) \end{array}$$

$$\begin{array}{lcl} \neg xt \wedge \neg xf & \rightarrow & \overline{ack}\uparrow \\ (xt \wedge x1) \vee (xf \wedge x0) & \rightarrow & \overline{ack}\downarrow \end{array}$$

In the design of the combinational expressions it is appreciated that this register contains both the true and inverted value as stable signals. For the simple output ports we have the production rules, Figure 5:

$$\begin{array}{lcl@{\qquad\qquad}lcl} x1 \wedge go & \rightarrow & \overline{yt}\downarrow & x0 \wedge go & \rightarrow & \overline{yf}\downarrow \\ \neg go & \rightarrow & \overline{yt}\uparrow & \neg go & \rightarrow & \overline{yf}\uparrow \end{array}$$

$$\begin{array}{lcll} \neg\overline{yf} & \rightarrow & \overline{yt}\uparrow & (\textit{staticizer}\,) \\ \neg\overline{yt} & \rightarrow & \overline{yf}\uparrow & (\textit{staticizer}\,) \end{array}$$

In the cases where combinational expressions are involved these has been designed by hand and incorporated in the output ports. The pull-down part of the production rule set for the $SUM$ function is:

$$\begin{array}{l} (\,(acc1 \wedge c0 \wedge (a0 \vee b0)) \vee (acc0 \wedge c1 \wedge (a0 \vee b0)) \vee \\ \quad (acc1 \wedge c1 \wedge (a1 \wedge b1)) \vee (acc0 \wedge c0 \wedge (a1 \wedge b1))\,) \wedge go \quad \rightarrow \quad \overline{sumt}\downarrow \end{array}$$

$$\begin{array}{l} (\,(acc0 \wedge c0 \wedge (a0 \vee b0)) \vee (acc1 \wedge c1 \wedge (a0 \vee b0)) \vee \\ \quad (acc0 \wedge c1 \wedge (a1 \wedge b1)) \vee (acc1 \wedge c0 \wedge (a1 \wedge b1))\,) \wedge go \quad \rightarrow \quad \overline{sumf}\downarrow \end{array}$$

These two expressions are manipulated and by transistor sharing the transistor count is brought down to 14 transistors for the two expressions (Figure 6).

The pull-down part of the production rules for $CARRY$ are:

$$\begin{array}{lcl} ((acc1 \wedge c1) \vee ((c1 \vee acc1) \wedge (a1 \wedge b1))) \wedge go & \rightarrow & \overline{carryt}\downarrow \\ ((acc0 \wedge c0) \vee ((c0 \vee acc0) \wedge (a0 \vee b0))) \wedge go & \rightarrow & \overline{carryf}\downarrow \end{array}$$

### Layout

The data path consists of seven registers and seven output ports (including those for assignments). The 218 transistors are laid out by hand, Figure 7.

## 7 Merge of control part and data path

We have implemented a prototype chip consisting of seven full adder cells plus the two end cells. In this design we decided to decompose the accumulation into single bit processors (section 2), which means that each bit process consists of a control part and a data path. The control signals ($NB$, $A$ and $R$) are distributed through a short tree structured fifo-queue. This structure ensures that the performance is independent of the length of the accumulator. For other implementations alternative trade-offs may be considered, for example one control unit for every four bits or only one for the whole accumulator. The drawback of the last solution is that the control signals has to be distributed to the whole array, the computation performed and the acknowledgement signals collected within the same cycle. This does not scale well.

# 8 Evaluation

An eight bit (4x4bit) multiply-accumulate unit has been fabricated as a prototype in $2\mu$ CMOS (MOSIS TinyChip service). The core of the chip measures $1830\text{x}1800\mu m$ and contains 3124 transistors. Each multiplier process consists of 347 transistors; 129 in the control part and 218 in the data path.

The chip is very robust towards variations in operating conditions. It has been tested successfully in the voltage range from below 0.8 volt to above 10 volts with repeated multiplication performance ranging from below 100 Kbit/sec to above 58 Mbit/sec. It should be noted that the accumulator is self-adjusting to these variations in operation conditions; it operates as fast as it can under the given conditions.

The performance at room temperature and 5 volts is:

| | |
|---|---|
| Multiplication with multiplier bit: | $t_a = 27$ nsec. $\approx 37$ Mbit/sec. |
| Input of new multiplicand: | $t_b = 37$ nsec. |
| Output of result: | $t_p = 30$ nsec. |

The implemented design is scalable to wider word sizes without loss of performance.

The accumulator operates with variable bit length of the multiplier with a performance for a multiply-accumulate operation in the range of 37 nsec to $n{\cdot}27$ nsec., where $n$ is the maximum size of the multiplier bit string.

## Acknowledgements

Our work with the serial-parallel multiplier originates from previous and ongoing work done in the VLSI group at the Technical University of Denmark.

We are indebted to Dražen Borković, Steve Burns, Marcel van der Goot, Pieter Hazewindus, Tony Lee and José Tierno, for patient guidance through the use of the tools and for their inspiring comments.

# References

[1] Steven M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits*, Ph.D.-thesis, Caltech, 1991.

[2] S.Y. Kung and J.N. Hwang. Parallel architectures for artificial neural nets. In *IEEE International Conference on Neural Networks*, volume 2, pages 165–172, 1988.

[3] Alain J. Martin. Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits. In C.A.R. Hoare, editor, *UT Year of Programming Institute on Concurrent Programming*, Addison-Wesley, 1989.

[4] Alain J. Martin. Synthesis of Asynchronous VLSI Circuits. In Jørgen Staunstrup, editor, *Formal Methods for VLSI Design*, North-Holland/Elsevier, 1990, pages 237–283.

[5] Alain J. Martin. Asynchronous Datapaths and the Design of an Asynchronous Adder. To appear in *Formal Methods in System Design*, Vol.1, nr. 1, Kluwer Academic Publishers, 1992.

[6] Christian D. Nielsen, Jørgen Staunstrup, and Simon Jones. A delay-insensitive neural network engine. In Will R. Moore, editor, *Proceedings of the Workshop on VLSI for Neural Networks*, September 1990.

[7] Christian D. Nielsen, Jørgen Staunstrup, and Simon Jones. Potential Performance Advantages of Delay Insensitivity. *IFIP Workshop on Silicon Architectures for Neural Nets*, September 1990.

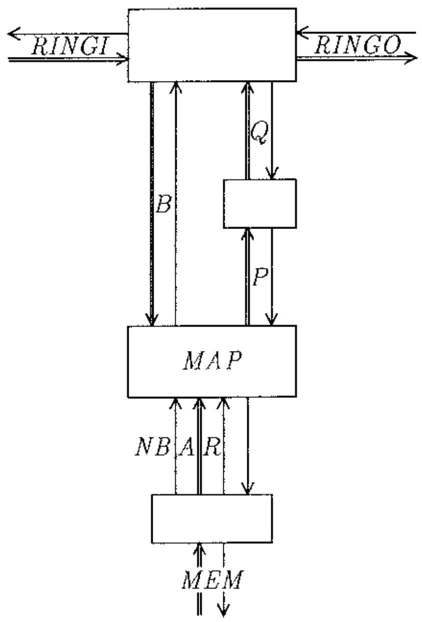

Figure 1: $MAP$ with environment processes and communication channels.

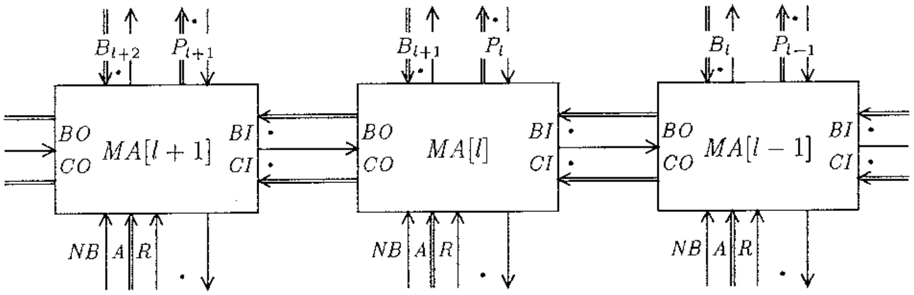

Figure 2: $MA[l]$ with connecting channels

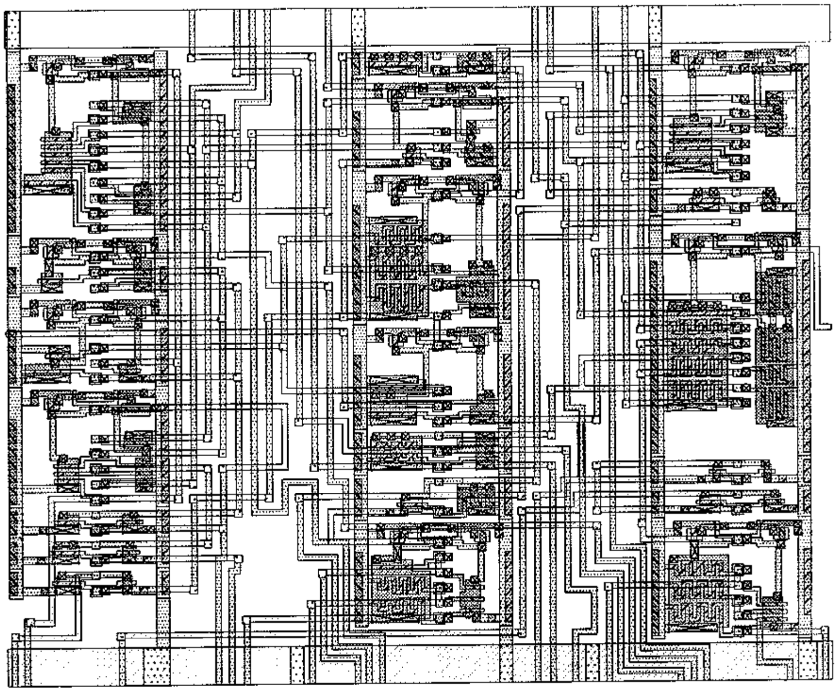

Figure 3: Layout for the control part.

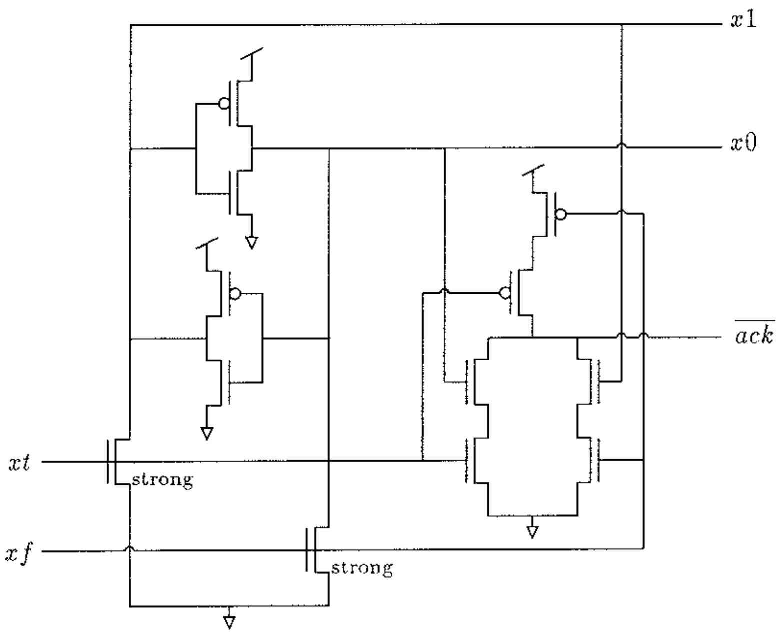

Figure 4: Transistor diagram for the standard register with acknowledgement

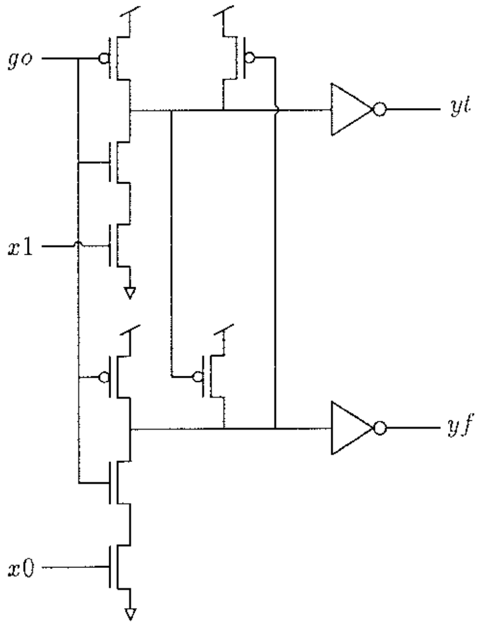

Figure 5: Transistor diagram for the standard output port

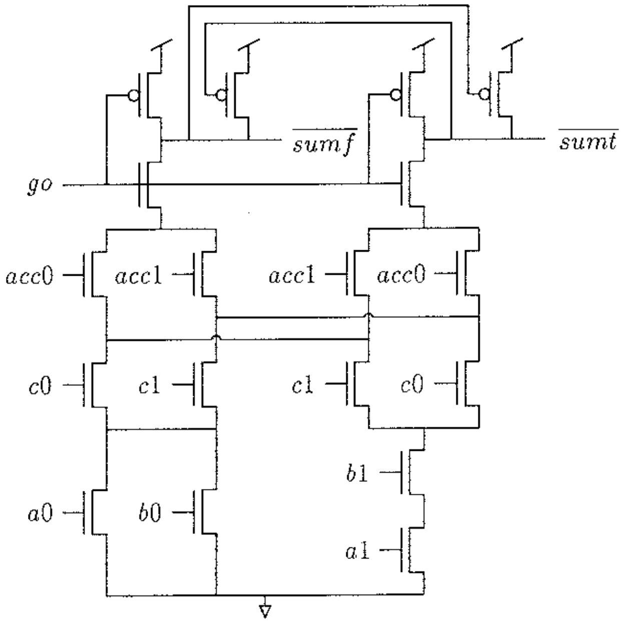

Figure 6: Transistor diagram for the *SUM*-function

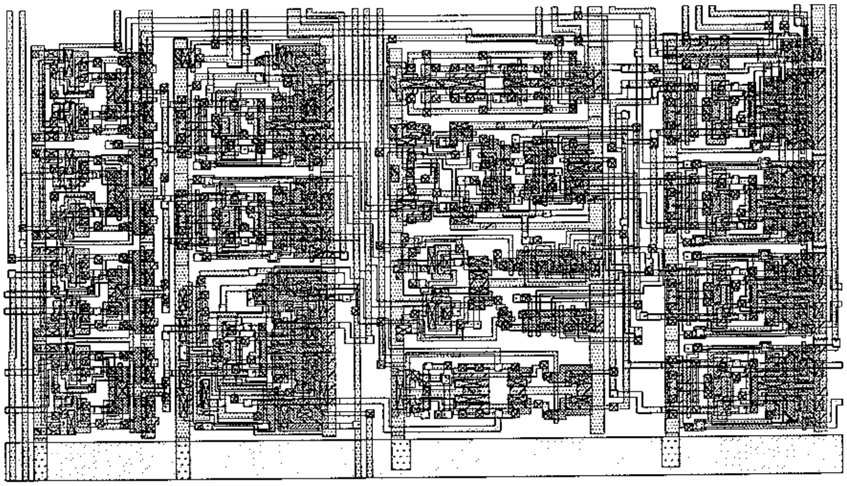

Figure 7: Layout for the datapath.